Gazi University Journal of Science
GU J Sci
26(1):85-95 (2013)

Machine Coded Genetic Algorithms For Real Parameter Optimization Problems

Mehmet Hakan SATMAN[1,♠]

[1]*Istanbul University, Department of Econometrics, Beyazit, Istanbul, TURKEY*

Received: 19.12.2011 Revised: 04.11.2012 Accepted: 24.11.2012

ABSTRACT

In this paper, we introduce a new encoding-decoding strategy for the floating-point genetic algorithms and we call the genetic algorithms which use this strategy Machine Coded Genetic Algorithms. We suggest applying classical crossover and mutation operations on the byte representations of real values which are already encoded in memory. This is equivalent to use a 256-unary alphabet with 8 genes for a single real value. Use of byte representations makes the classical genetic operators interpretable in floating-point chromosomes and increases the search capabilities in a wide range without losing accuracy. This strategy also decreases the computation time needed for the genetic operators. Simulation studies show that our strategy performs well on many test functions by means of converging to global optimum and time efficiency.

Key Words: *Genetic algorithms, Chromosome encoding, Real parameter optimization.*

1. INTRODUCTION

The classical genetic algorithm (GA) is based on binary coded chromosomes (Holland, 1975), (Goldberg, 1989). Since GAs run with the fitness values of candidate solutions rather than the goal function itself, they are problem independent. They do not require the functions to be continuous and differentiable and they do not suffer

♠Corresponding author, e-mail: mhsatman@istanbul.edu.tr from local optima when the population size is large enough and the initial population is well randomized to obtain diversity.

Although the GA performs well on many problems, it is more efficient to use floating-point genetic algorithms (FPGAs) in real-valued problems. Janikow and Michalewicz (1991) stated that the FPGA is faster, more consistent and precise than the GA. Binary chromosomes need more bits to represent a wider solution space whereas the use of floating-point parameters makes it possible to use larger domains even the limits of parameters are unknown (Herrera et al., 1998). However, crossover and mutation operations do not find a place in FPGA's at first glance. The mutation operator which is defined as flipping a single bit is not meaningful for real numbers. It is also not clear how to combine two parent real vectors to produce new real vectors (Deb, 2004).

Radcliffe (1992) argued that there is no requirement that the idealized genetic operators used in evolutionary search be defined with respect to the chosen representation, nor indeed with respect to any specific representation. In addition to this, different optimization algorithms with different coding-decoding strategies can either be successful with different configurations, which are to say, the success of genetic algorithms is not related to coding-decoding strategy (Herrera et al., 1998) (Fogel and Ghozeil, 1997). Finally, Reeves (1993) showed that a higher cardinality alphabet needs the population size to be larger in order to ensure diversity.

In this paper, we introduce a new encoding-decoding strategy for FPGAs. In our algorithm, chromosomes are real values but the crossover and the mutation operations are applied on byte representations of chromosomes in computer memory so that it neither encodes chromosomes nor uses real values in genetic operations. The byte representation is similar to binary representation so the original interpretation of genetic operators is clearer and finds a place in FPGAs. A C program is written as an implementation and the R (R Development Core Team, 2012) wrapper package *mcga* (Satman, 2012) is written and ready for downloading at the CRAN repositories. The source code of the C library is included in this package and ready for use in both languages. Note that simulations on this paper are performed using the R package. External function calls may slow down the library but the interactive R console may be helpful to show the search capabilities of MCGA.

In Section 2, we discuss the use of some genetic operators in FPGAs. In Section 3, we define a new hybrid strategy for crossover and mutation operations and introduce the machine coded genetic algorithms (MCGAs). In Section 4, we give a brief description of our implementation. In Section 5, we prepare a simulation study to compare the performance of MCGA with some other evolutionary algorithms using a test case of well-known functions.

2. OPERATORS IN FLOATING-POINT GENETIC ALGORITHMS

There are many reasons for using FPGAs in real-valued optimization problems. Chromosomes are real vectors and there is no need for long bit strings. Each single gene of a chromosome is a real number which has high precision in a wide range. There is also some attempt to make GAs more precise with a limited chromosome length in real-valued problems. To ensure precision in the GAs, Schraudolph and Belew (1992) suggested the method of Dynamic Parameter Encoding which is based on a zooming operator. Their algorithm starts the search in a predefined range and narrows this range after many generations in order to represent more precise numbers with the same length of bit strings.

As we mentioned in Section 1, there is no clear equivalent method for the classical crossover and the mutation operations in FPGAs. Some authors suggested many methods for those operators whereas the classical mutation has a unique definition which is simply flipping a bit and the classical crossover has similar definitions which are aimed to combine two chromosomes. Suppose that c^1 and c^2 are chromosomes of a population P with binary content. The one-point crossover operation of GAs can be applied in a single point as

$$c^3 = c_1^1, c_2^1, \dots, c_{h-1}^1, c_h^2, c_{h+1}^2, \dots, c_k^2 \qquad (1)$$

where k is the chromosome length and h is the cut point where $0 < h < k$. One can apply this naive method on the floating-point chromosomes. But it is clear that this method combines variables rather than assembles piece of blocks. Deb (2004) argued that this approach can-not diversify the candidates and mutation attains more importance. Note that choosing more than one cut points is possible but its task will still be far from being fulfilled.

A better approach is producing offspring as linear combinations of parent chromosomes. Suppose that f_1 and f_2 are floating-point chromosomes which contain real values. The linear combination of those parents can be written as

$$f_3 = \alpha f_1 + (1 - \alpha) f_2 \qquad (2)$$

where α is a real value. When α is selected within the range $0 < \alpha < 1$, this operator is equivalent to weighted average of two parents. Other values for α, for example 1,5, might be used to generate new offspring that lie out of the space spanned by the parents.

There are many crossover methods in the literature and almost all of them are based on linear or non-linear combinations of real values which are contained by floating-point chromosomes. Deb (2004) concluded that crossover methods developed for FPGAs have equal performance and are content dependent. Elsayed et al. (2011) performed a simulation study among the crossover and mutation methods in constraint problems and concluded that there is no superior method. Herrera et al. (2003) made a comprehensive and comparative study among the crossover methods in FPGAs and concluded that additional study for developing new methods is necessary in this area.

Mutation operators in FPGAs also differ from GAs and are generally based on adding a value from a user defined range or a preselected scheme. Fine tuning is the key but in most of the problems, specifying the configuration is more difficult than solving the problem.

3. MACHINE-CODED GENETIC ALGORITHMS

In computer programs, generally compiled ones, numerical data are stored as byte arrays in the memory. A byte is a union of eight-bits and each single bit can take a value of zero or one. Few bytes are required if the stored number is small in digits. But if the number has more digits or precision is important more bytes are required. This is why data types were implemented in compilers and interpreters.

The number of bytes used for storing a numerical value in the memory is finite, that means, there is no way for representing real-values with exact precision. As the number of bytes increases, precision also increases and real values can be represented in a reasonable form.

One of the most popular programming languages, C, has numerous data types for storing or pointing a numerical value. Suppose that p is a double-precision variable with the value of π with 15 decimal points. The definition of variable p can be written as

```
double p = 3.141592653589793;
```

If the compiler generates 32-bit code, variable p is 8 bytes long, which can be proven using the expression *sizeof*(*double*). The byte representation of p can be obtained using the C code

```
unsigned char *cpp = (unsigned char*) p;
```

and is shown in Table 1.

Table 1. Byte representation of variable *p*.

	1	2	3	4	5	6	7	8
Bytes	24	45	68	84	251	33	9	64

The byte array shown in Table 1 is a result of a formulation algorithm defined in IEEE 754 - IEEE Standard for Floating-Point Arithmetic (Stevenson, 1981). Compilers mostly use the same standard for converting floating-point numbers and byte arrays each other, which is to say, same result should be obtained in Java.

Each single byte represented in Table 1 has a different effect on the variable p. In Table 2, partial effects are shown when the value of a single byte is increased and decreased by 1.

Table 2. Partial effects of changing the values of bytes of *p*.

Value	1	2	3	4	5	6	7	8
0.000047936899621	24	45	68	84	251	33	9	**63**
3.**016592653589793**	24	45	68	84	251	33	**8**	64
3.141**104372339793**	24	45	68	84	251	**32**	9	64
3.14159**0746241160**	24	45	68	84	**250**	33	9	64
3.1415926**46139213**	24	45	68	**83**	251	33	9	64
3.1415926535**60689**	24	45	**67**	84	251	33	9	64
3.141592653589**679**	24	**44**	68	84	251	33	9	64
3.141592653589793	**23**	45	68	84	251	33	9	64
3.141592653589793	24	45	68	84	251	33	9	64
3.14159265358979**4**	**25**	45	68	84	251	33	9	64
3.141592653589**907**	24	**46**	68	84	251	33	9	64
3.141592653**618897**	24	45	**69**	84	251	33	9	64
3.1415926**61040374**	24	45	68	**85**	251	33	9	64
3.14159**4560938426**	24	45	68	84	**252**	33	9	64
3.14**2080934839793**	24	45	68	84	251	**34**	9	64
3.**266592653589793**	24	45	68	84	251	33	**10**	64
205887.41614566...	24	45	68	84	251	33	9	**65**

In Table 2, it is shown that, a small change in a byte leads to a small change in the value of p if the byte is in the left side of the array, while a small change in the right most byte leads to a higher effect. This is the mutation operator of MCGA, it is based on changing a byte by 1 rather than adding a random value.

Effect of the mutation operation depends on the location of the mutated byte. This is similar with the mutation operator of binary-coded genetic algorithms. In MCGA, each byte of a chromosome is mutated with probability P_M. If a byte is subject to be mutated it is increased by +1 or -1 with probability ½.

The main advantage of using such an operator is obtaining extremely small and huge changes without fine tuning of optimization and operator parameters.

The crossover operation is also performed using the byte array representations of chromosomes. Suppose that e is a double variable with the value of $exp(1)$ with 15 decimal points. The C definition of this variable is shown in the code below:

```
double e = 2.718281828459045;

unsigned char *cpe = (unsigned char*) e;
```

Results of the one-point crossover and the uniform crossover are shown in Table 3 and Table 4, respectively. It can be seen that crossover operations produce new values that must not lie between two parents. But it can be said that produced values obtained after crossover operation are not far away from the parents.

Table 3. One-point crossover on different cut-points.

Value	1	2	3	4	5	6	7	8
3.141592653589793	24	45	68	84	251	33	9	64
2.718281828459045	**105**	**87**	**20**	**139**	**10**	**191**	**5**	**64**
2.718281420069285	24	45	68	84	**10**	**191**	**5**	**64**
3.141593061979553	**105**	**87**	**20**	**139**	251	33	9	64
2.641592653589793	24	45	68	84	251	33	**5**	**64**
3.218281828459045	**105**	**87**	**20**	**139**	**10**	**191**	9	64

Table 4. Uniform crossover.

Value	1	2	3	4	5	6	7	8
3.141592653589793	24	45	68	84	251	33	9	64
2.718281828459045	**105**	**87**	**20**	**139**	**10**	**191**	**5**	**64**
2.641593061974742	24	45	**20**	**139**	251	33	**5**	**64**
3.218281420074096	**105**	**87**	68	84	**10**	**191**	9	64
3.218741500876501	24	**87**	68	**139**	251	**191**	9	**64**
2.641132981172337	**105**	45	**20**	84	**10**	33	**5**	64

Suppose that the offspring in Table 3 and Table 4 are generated using the formula (2) where f_1 and f_2 are floating-point chromosomes with values d and e. One can find the values of alphas for each single offspring using the formula

$$\alpha = \frac{g-e}{d-e} \quad (4)$$

where g is the generated content which is the result of the arithmetic crossover operation. Note that there is not a single α and this operation is equivalent to (2) when the α is selected randomly in each operation.

This operator simply does what the crossover operator does in GAs. It uses building blocks of chromosomes and assemblies them rather than using the real values directly. Its main advantage is being fast, which is to say, it does not require performing arithmetic operations, especially multiplication and division, which consume time.

MCGA uses tournament selection as selection scheme. Tournament selection is a non-parametric method which does not need extra transformations performed on objective values. In tournament selection, k objective function values (or fitness values) are selected randomly from the population and the winner is labeled as $winner_1$ where $k \leq n$ and n is the population size. Then k objective function values are selected randomly and the best one is labeled as $winner_2$. In MCGA, k is set to 2.

The best solution in a population is always labeled as winner if it is selected for a tournament and the worst one loses all tournaments. Tournament selection has better or equivalent convergence and computational time complexity properties when compared to other selection operators that exists in the literature (Goldberg and Deb, 1991) (Deb, 2004: 89). Since MCGA uses tournament selection, objective values returned by objective functions are directly comparable for determining the winners.

Steps of MCGA can be listed in Algorithm 1.

Algorithm 1. Steps of MCGA

Step-0) Define the goal function. Goal function is a function which takes candidate solutions as input and returns a cost value as output. Type of the problem is minimization, by default. Define population size *PopSize, c*rossover probability P_C, mutation probability P_M and elitism parameter *e*.

Step-1) Construct a random population and an empty population with sizes of *PopSize*. The former is labeled as current population and the latter is labeled as next population. Chromosome length is equal to number of inputs defined in goal function.

Step-2) Calculate cost values, copy the best *e* solutions into next population.

Step-3) Apply tournament selection. Apply crossover operator on byte representations of selected chromosomes with probability P_C.

Step-4) Apply mutation operator on byte representations of offspring generated in Step-3 with probability P_M. Copy generated offspring into next population. If *PopSize* chromosomes are copied into next population go to Step-5 else go to Step-3.

Step-5) Swap the current and the next populations. Check if the current number of iterations is equal to maximum number of iterations. If not, go to Step-2 else go to Step-6

Step-6) Sort the population by cost values in ascending order. Report the first chromosome as the final solution.

4. C AND R IMPLEMENTATIONS OF MCGA

MCGA is implemented in C, one of the most popular programming languages. Easiness of pointing memory and type casting make C the most proper language for such an encoding-decoding strategy. Note that, implementation is possible with other languages, but would be slow or impossible if the compiler or interpreter denies direct access to memory. On the contrary, direct access to memory is denied in Java but floating-point variables and byte arrays can be converted to each other by using *ByteArrayInputStream* , *ByteArrayOutputStream*, *ObjectInputStream* and *ObjectOutputStream* classes.

Table 5. Parameters of *mcga.*

mcga(popsize, chsize, crossprob = 1, mutateprob = 1/100, elitism = 1, minval, maxval, maxiter = 10, evalFunc)	
popsize	Number of chromosomes.
chsize	Number of parameters.
crossprob	Crossover probability. By default it is 1
mutateprob	Mutation probability. By default it is 1/100
elitism	Number of best chromosomes to be copied into next generation. By default it is 1
minval	Lower bound of the randomized initial population.
maxval	Upper bound of the randomized initial population.
maxiter	Maximum number of generations. By default it is 10
evalFunc	An R function. By default, each problem is a minimization.

The R package, *mcga*, is written in R to wrap the original C code. Having an interactive console and easiness of calling compiled code in an interpreted manner make R proper for our testing issues. Parameters and their descriptions of *mcga* are given in Table 5.

After a function call, *mcga* returns a list containing a matrix of the final population and a vector of the corresponding costs. Members of the final population are sorted by corresponding cost values.

5. SIMULATION STUDY

We perform a simulation study to compare search capabilities and time efficiency of MCGA with some well-known evolutionary algorithms. Differential evolution (Storn and Price, 1997) is an other evolutionary algorithm in this subject. In their simulation studies, Vesterstrøm and Thomsen (2004) showed that differential evolution (DE) is the best algorithm among others including particle swarm optimization (Poli et al., 2007). Covariance matrix adaptation evolution strategy (CMA-

ES) is an other successful method developed for real-valued optimization problems. Hansen and Kern (2004) show that CMA-ES performs well on a set of test functions with different number of parameters from 2 to 80.

In our simulation study, we compare MCGA with DE, CMA-ES and FPGA using their implementations *mcga*, *DEoptim* (Mullen et al., 2011), *cmaes* (Trautmann et al., 2011) and *genalg* (Willighagen, 2005), respectively. We run our simulations on the eight test functions with number of parameters $p = 2, 5, 10, 20, 40, 80$. This test suite is same of the previous work reported by Hansen and Kern (2004) and shown in Table 6. However, we use a limited configuration in our simulations. For convenience, we set the population size and the maximum number of generations to 100. So the number of maximum function evaluations is 10000 for all. We apply the crossover operation for all selected chromosomes in MCGA and FPGA, so the crossover probability is 1. The probability of mutating a single gene is set to 0,05 and the best chromosome is directly copied to next generation for those algorithms. All of the test functions have a known global minimum of 0 for all $\boldsymbol{x} = 0$ except Schwefel has the global minimum of 420,96874636. Simulations are performed 100 times for each single configuration. Results of the simulation study are shown in Figure 1 and Figure 2.

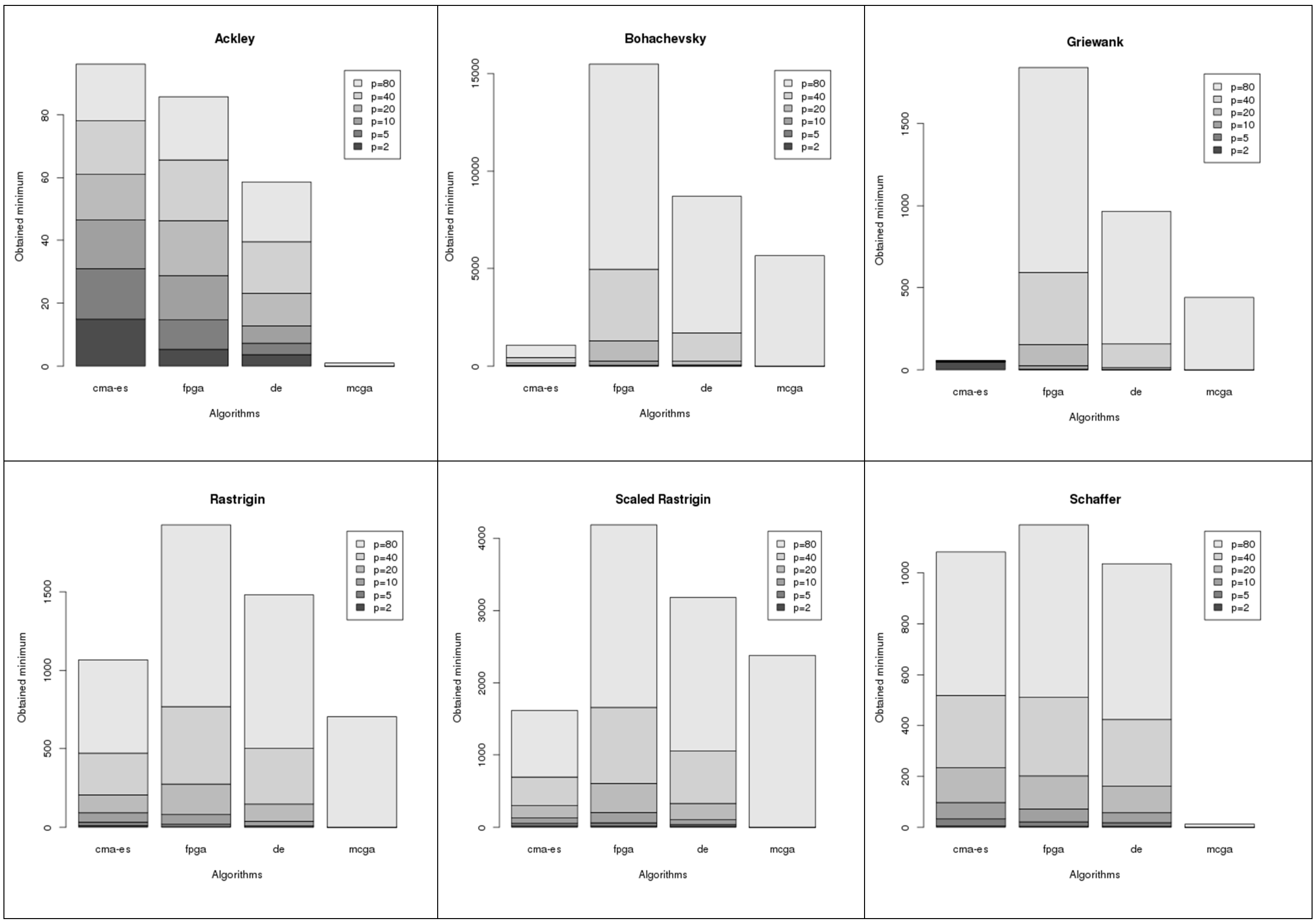
As shown in Figure 1, CMA-ES and MCGA are prominent methods as they have smaller objective function values in average. CMA-ES has the worst performance on Ackley, while the average of minimum values obtained by MCGA is close to zero in all configurations. MCGA outperforms CMA-ES for $p \neq 80$ on functions Bohachevsky, Griewank, Rastrigin, Scaled Rastrigin and Skew Rastrigin. However, CMA-ES converges better on those algorithms when $p = 80$. In addition to this, MCGA outperforms CMA-ES for all p values when the objective function is Schaffer. Beyond this nice picture, MCGA is outperformed by other algorithms on Schwefel. Since, the objective function is defined in the range of $[-500{,}300]^n$, initially randomized population of MCGA possibly includes both positive and negative candidates. This doubles the search space and reaching the global minimum requires more iterations and chromosomes.

Table 6. Test Functions (Hansen and Kern, 2004).

Name	Function	Init
Ackley	$f(x) = 20 - 20\exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}) + e - \exp(\frac{1}{n}\sum_{i=1}^{n} \cos(2\pi x_i))$	$[1{,}30]^n$
Bohachevsky	$f(x) = \sum_{i=1}^{n-1}(x_i^2 + 2x_{i+1}^2 - 0.3\cos(3\pi x_i) - 0.4\cos(4\pi x_{i+1}) + 0.7)$	$[1{,}15]^n$
Griewank	$f(x) = \frac{1}{4000}\sum_{i}^{n} x_i^2 - \prod_{i}^{n} \cos(\frac{x_i}{\sqrt{i}}) + 1$	$[10{,}600]^n$
Rastrigin	$f(x) = 10n + \sum_{i=1}^{n}(x_i^2 - 10\cos(2\pi x_i))$	$[1{,}5]^n$
Scaled Rastrigin	$f(x) = 10n + \sum_{i=1}^{n}((10^{\frac{i-1}{n-1}}x_i)^2 - 10\cos(2\pi 10^{\frac{i-1}{n-1}}x_i))$	$[1{,}5]^n$
Schaffer	$f(x) = \sum_{i=1}^{n-1} (x_i^2 + x_{i+1}^2)^{0.25}[\sin^2(50(x_i^2 + x_{i+1}^2)^{0.1}) + 1.0]$	$[10{,}100]^n$
Schwefel	$f(x) = 418.9828872724339n - \sum_{i=1}^{n} x_i\sin(\sqrt{\lvert x_i\rvert})$	$[-500{,}300]^n$

Skew Rastrigin	$f(x) = 10n + \sum_{i=1}^{n}(y_i^2 - 10\cos(2\pi y_i))$ $where y_i = \begin{cases} 10x_i & , \quad x_i > 0 \\ x_i & , \quad otherwise \end{cases}$	$[1,5]^n$

As shown in Figure 2, CMA-ES and FPGA are outperformed by DE and MCGA by means of time efficiency. CPU times in seconds are obtained using an Intel i5 machine with 4 Gb memory installed. The C library is compiled with GCC on Linux (Ubuntu) operating system. Although the average times represented in Figure 2 seem similar for DE and MCGA, differences between central tendencies of times are significant for many cases. We test the null hypothesis $H_0: \mu_{Time(MCGA)} \leq \mu_{Time(DE)}$ with the alternative hypothesis $H_a: \mu_{Time(MCGA)} > \mu_{Time(DE)}$ using Wilcoxon rank-sum test where $\mu_{Time(MCGA)}$ and $\mu_{Time(DE)}$ are location parameters of calculation times for MCGA and DE, respectively.



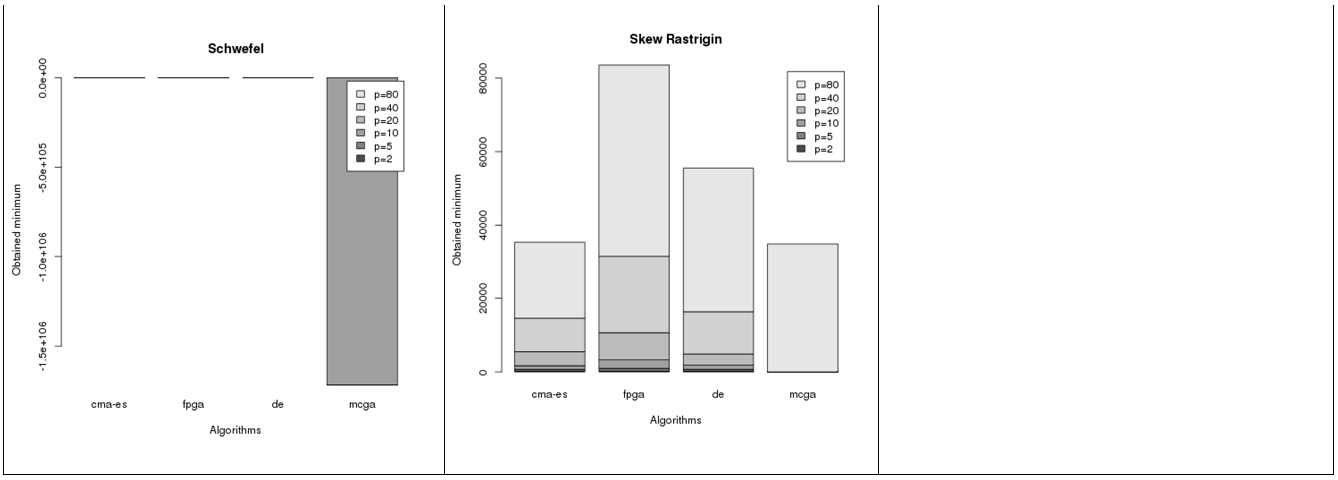
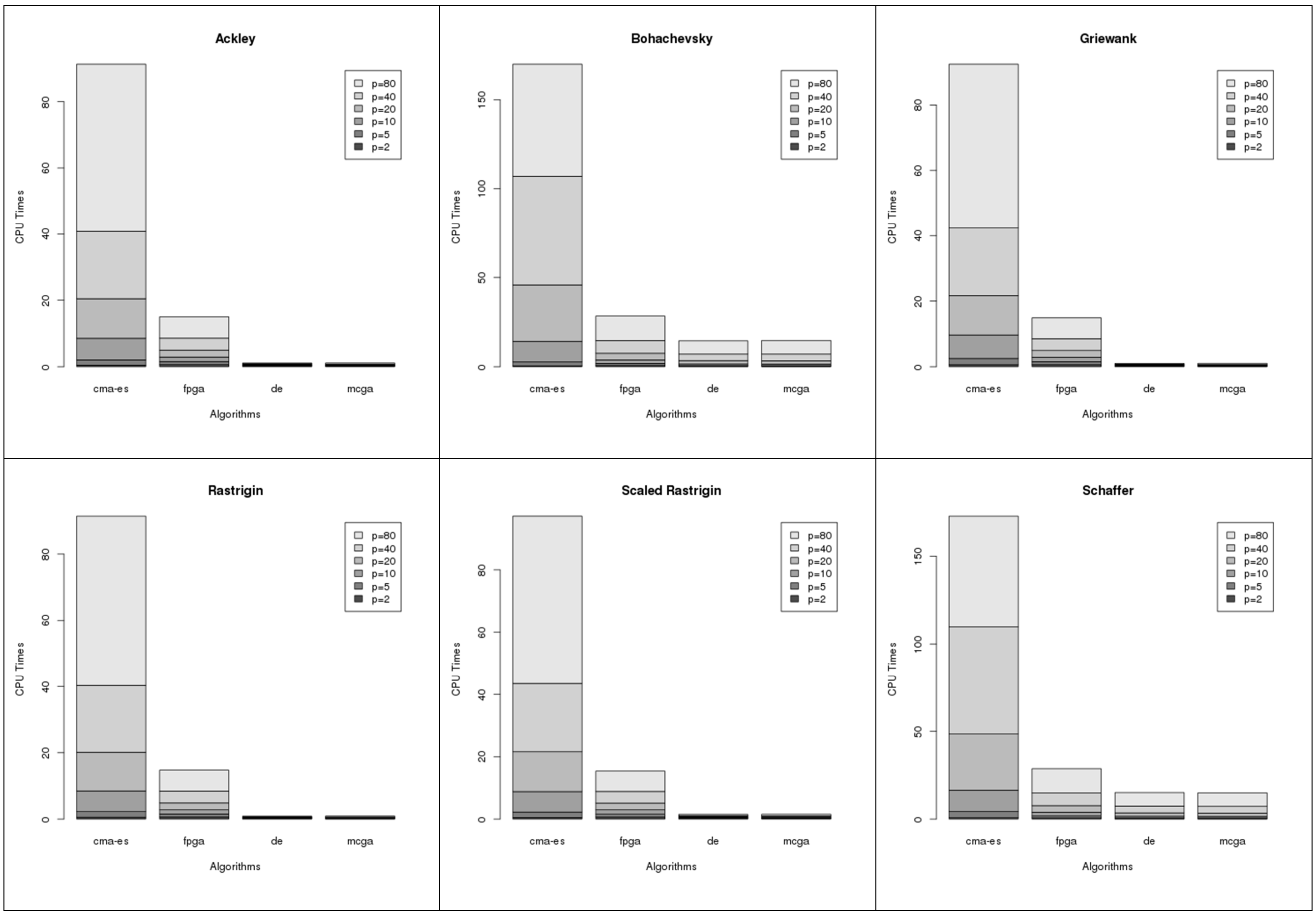


Figure 1. Test functions and average of minimum values obtained by algorithms.

We fail to reject the null hypothesis for all test functions in all dimensions except $p = 40$ and $p = 80$ using the significance level $\alpha = 0{,}05$. In the cases of $p = 40$ and $p = 80$, DE outperforms MCGA on all functions except Schaffer and Skew Rastrigin. Note that, those performance reports are related to language differences, that are, CMA-ES and FPGA are written in R which is an interpreted language while DE and MCGA are compiled to machine code and wrapped by R functions. However, test functions given in Table 6 are implemented in R. This strategy standardizes the time consumed by function evaluations, which spans the major portion of consumed time.



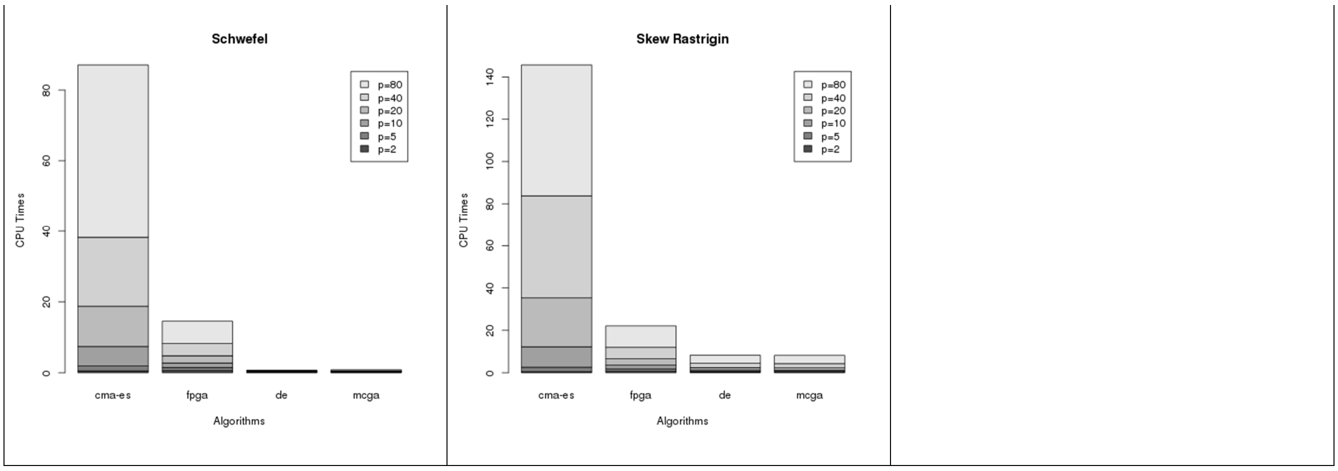
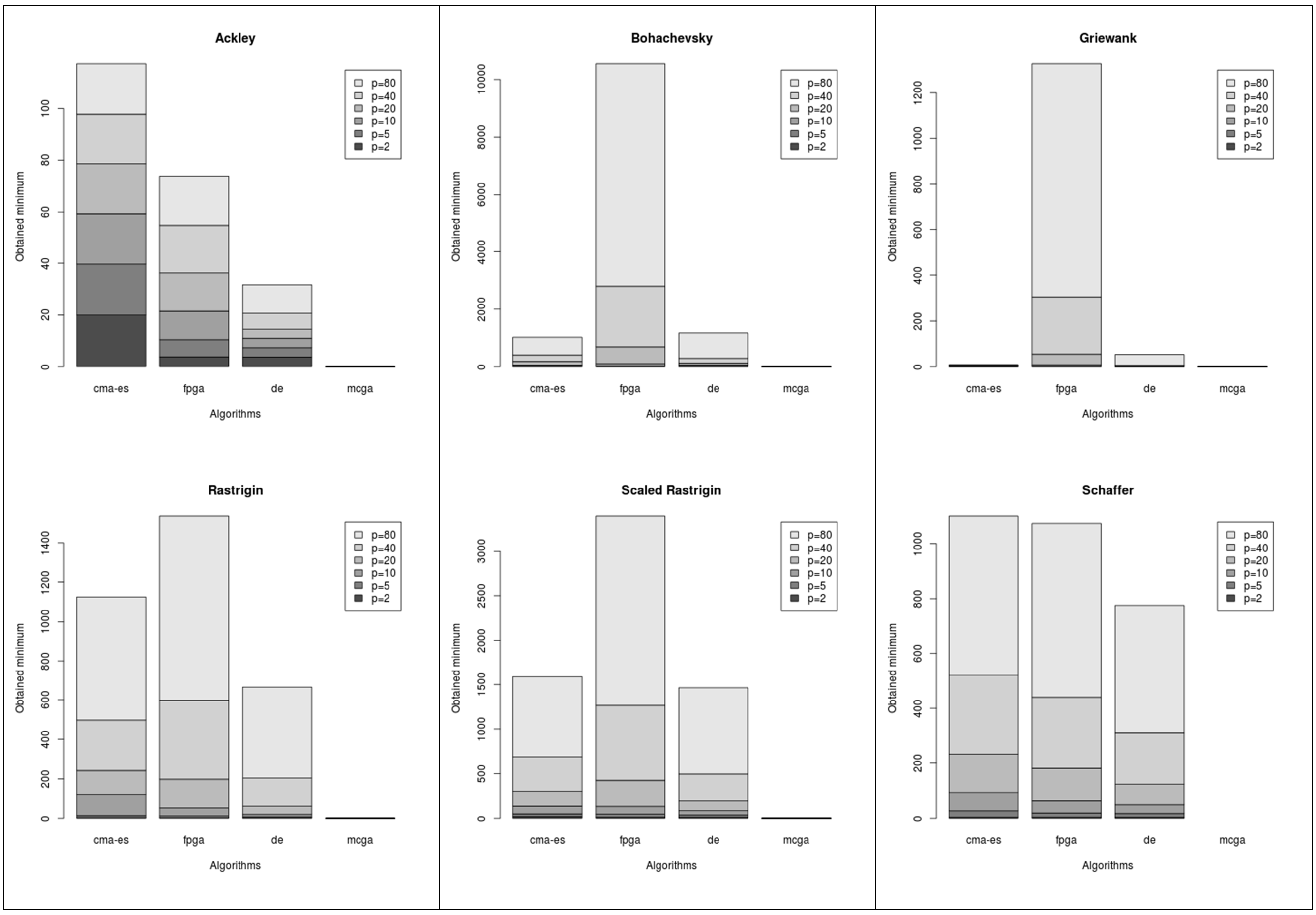


Figure 2. Performance of algorithms in CPU times.

We perform the same simulation study to compare search capabilities of algorithms with more iterations. We set the number of iterations to 500 and results of this simulation are given in Figure 3.



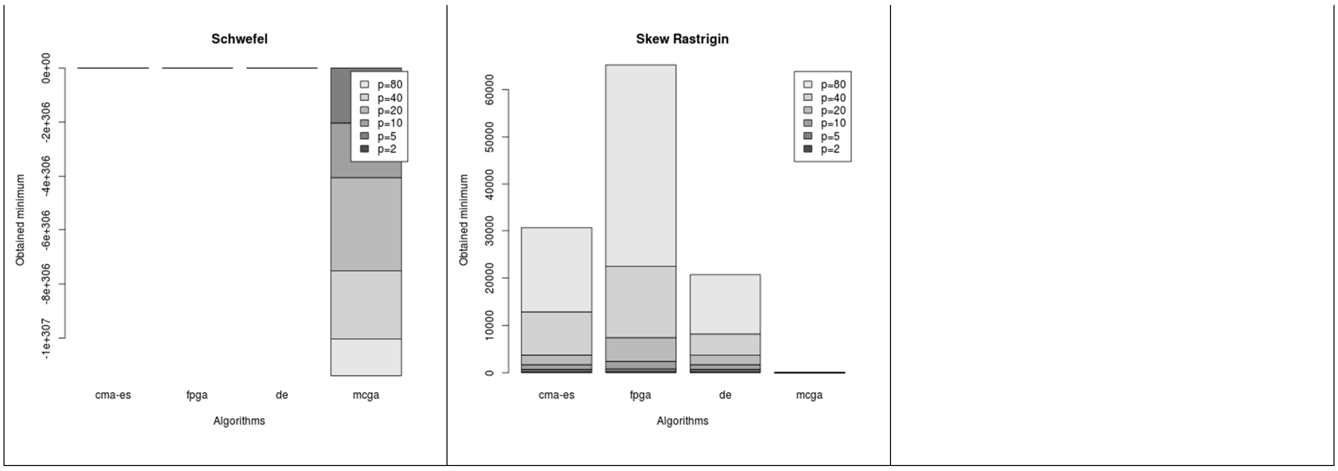


Figure 3. Test functions and average of minimum values obtained by algorithms with more iterations.

In Figure 3, it is shown that MCGA outperforms the other algorithms in all dimensions and all functions except Schwefel. It is also shown that, MCGA obtains smaller objective values when the number of iterations is 500 in higher dimensions. CMA-ES and DE have reasonable convergence properties relative to FPGA. Comparisons of time efficiencies are nearly same as in Figure 2. DE and MCGA are the prominent algorithms by means of time efficiency.

6. CONCLUSION

In this paper, we suggest a new encoding-decoding strategy for the floating-point chromosomes. Computers already encode floating-point numbers using a high order alphabet which is called a byte. Performing the classical crossover and the mutation operations on those bytes makes these operators interpretable on the floating-point genetic algorithms. Use of these suggested operators also reduces the effort for the fine tuning of the algorithm parameters and CPU time required by genetic operators which are mostly based on multiplication and division. Floating-point variable types are designed to store an huge interval of numbers and the genetic operators suggested in this paper are capable to explore the entire search space in reasonable times. We perform a simulation study to compare performances of our suggested algorithm and some other evolutionary algorithms by means of convergence property and time efficiency. Results of our simulation study show that MCGA converges faster when the number of dimensions is relatively small even with a limited configuration. It is also shown that MCGA reaches the global optimum in higher dimensions for most of test functions when the number of iterations is moderate.

REFERENCES

[1] Deb, K., “Multi-Objective Optimization using Evolutionary Algorithms”, ***John Wiley & Sons***, (2004).

[2] Elsayed, S.M., Sarker, R.A., Essam, D.L., “Multi-operator Based Evolutionary Algorithms for Solving Constrained Optimization Problems”, ***Computers & Operations Research***, 38: 1877-1896 (2011).

[3] Fogel, D.B., Ghozeil, A., “A Note on Representations and Variation Operators”, ***IEEE Transactions on Evolutionary Computation***, 1: 2, July (1997).

[4] Goldberg, D., “Genetic Algorithms in Search”, ***Optimization, and Machine Learning***, Addison-Wesley, (1989).

[5] Goldberg, D. E., Deb, K., “A Comparison of selection schemes used in genetic algorithms”, ***In Foundations of Genetic Algorithms 1 (FOGA-1)***, 69-93 (1991).

[6] Herrera, F., Lozano, M., Sanchez, A.M., “A Taxonomy for the Crossover Operator for Real-Coded Genetic Algorithms: An Experimental Study”, ***International Journal of Intelligent Systems***, 18: 309-338 (2003).

[7] Herrera, F., Lozano, M., Verdegay, J.L., “Tackling Real-Coded Genetic Algorithms: Operators and Tools for Behavioural Analysis”, ***Artificial Intelligence Review***, 12: 265-319 (1998).

[8] Hansen, N., Kern, S., “Evaluating the CMA Evolution Strategy on Multimodal Test Functions”, ***Parallel Problem Solving from Nature - PPSN VIII***, 282-291, (2004).

[9] Holland, J.H.,” Adaptation in Natural and Artificial Systems”, University of Michigan Press, (1975).

[10] Janikow, C.Z., Michalewicz, Z., “An Experimental Comparison of Binary and Floating Point Representations in Genetic Algorithms”, ***Proceedings of the Fourth International Conference on Genetic Algorithms***, Morgan Kaufmann, 31-36 (1991).

[11] Mullen, K., Ardia, D., Gil, D., Windover, D., Cline, J., “DEoptim: An R Package for Global Optimization by Differential Evolution”, ***Journal of Statistical Software***, 40(6): 1-26 (2011).

[12] Poli, R., Kennedy, J., Blackwell, T., “Particle swarm optimization - An overview, ***Swarm Intelligence***, 1: 33-57 (2007).

[13] R Development Core Team, “R: A Language and Environment for Statistical Computing”, ***R Foundation for Statistical Computing***, *http://www.R-project.org/*, (2012).

[14] Radcliffe, N.J., “Non-Linear Genetic Representations”, ***Parallel Problem Solving from Nature 2***, R. Manner and B. Manderick (Ed.) (Elsevier Science Publishers, Amsterdam), 259-268, (1992).

[15] Reeves, C.R., “Using Genetic Algorithms with Small Populations”, ***Proceedings of the 5th International Conference on Genetic Algorithms***, Morgan Kaufmann, San Mateo, CA, (1993).

[16] Satman, M.H., “mcga: Machine coded genetic algorithms for real-valued optimization problems”, ***R package version 2.0.6***, http://cran.r-project.org/web/packages/mcga/index.html, (2012).

[17] Schraudolph, N.N., Belew, R.K., “Dynamic Parameter Encoding for Genetic Algorithms”, ***Machine Learning***, 9: 9-21 (1992).

[18] Stevenson, D., “A Proposed Standard for Binary Floating-Point Arithmetic”, ***Draft 8.0 of IEEE Task P754***, 10.1109/C-M.1981.220377, 51-62 (1981).

[19] Storn, R., Price, K., “Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces” , ***Journal of Global Optimization***, 11: 341-359 (1997).

[20] Trautmann, H., Mersmann, O., Arnu, D., “cmaes: Covariance Matrix Adapting Evolutionary Strategy”, ***R package version 1.0-11***, http://CRAN.R-project.org/package=cmaes, (2011).

[21] Vesterstrøm, J., Thomsen, R., “A Comparative Study of Differential Evolution, Particle Swarm Optimization, and Evolutionary Algorithms on Numerical Benchmark Problems” , ***CEC2004. Congress on*** June, 2: 1980-1987 (2004).

[22] Willighagen, E., “genalg: R Based Genetic Algorithm”, ***R package version 0.1.1***, http://cran.r-project.org/web/packages/genalg/index.html, (2005).